2INC0 - Operating Systems

Processes

Dr. Geoffrey Nelissen

Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2 and 3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - action synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)
- **File systems** (lectures 8)
- **Memory management** (lectures 9 and 10)
- **Input/output** (lecture11)

Announcement

- **First practical assignment** will be **released today**.
  - **Deadline in 2.5 weeks**
  - There is a question in the assignment asking to **show that the program may reach a deadlock**. This **topic is covered next Friday**.
  - Possibility to get a **bonus point** by submitting a second solution avoiding the deadlock situation.

- **First homework** assignment will be **released today.**
  Deadline Sunday of next week.
  - We redesigned the homework based on previous years students' feedback.
  - Homework requires to solve exercises similar to those done in today's lecture.

- Check that you are **registered to a group** for the practical assignments (groups of 3 students)

Agenda

- Reminder of lecture 1 (quiz)
- Processes

Reminder of lecture 1

- Connect to www.menti.com
- Code 7553 5681

Question 1

Which of the following is (are) motivation(s) for the existence of operating systems?

1. **Improve portability**
2. **Manage concurrency**
3. **Reduce execution overhead**
4. **Provide a simplified view of the execution platform**
5. **Provide support for shared functionality**
6. **Specify a unified hardware specification**
7. **Accelerate memory and I/O accesses**

Connect to www.menti.com
Code 7553 5681

Question 1

Which of the following is (are) motivation(s) for the existence of operating systems?

1. Improves portability
2. Manages concurrency
3. Reduces execution overhead
4. Provides a simplified view of the execution platform
5. Provides support for shared functionality
6. Specifies a unified hardware specification
7. Accelerates memory and I/O accesses

Connect to www.menti.com
Code 5402 5022

(green = correct, red = incorrect)

Question 2

TU/e

Which of the following is/are true about DMA (Direct Memory Access)

1. **A DMA is a memory access protocol**
2. **A DMA virtualizes memory**
3. **A DMA is part of the operating system**
4. **A DMA replaces I/O controllers**
5. **A DMA is a hardware component**

IRiS

Connect to www.menti.com
Code 7553 5681

Question 2

Which of the following is/are true about DMA (Direct Memory Access)

1. A DMA is a memory access protocol
2. A DMA virtualizes memory
3. A DMA is part of the operating system
4. A DMA replaces I/O controllers
5. A DMA is a hardware component

(green = correct, red = incorrect)

Connect to www.menti.com
Code 5402 5022

Question 3

Which of the following is/are true?

1. A DMA increases the work done by the CPU

2. A DMA may impact negatively the execution time of a program

Connect to www.menti.com
Code 7553 5681

Question 3

Which of the following is/are true?

1. A DMA increases the work done by the CPU
2. A DMA may impact negatively the execution time of a program

Question 5

Which of the following is/are true about interrupts and traps?

1. **Upon an interrupt, the CPU enters into kernel mode.**
2. **An interrupt is a signal from the CPU to a device controller, informing it that the CPU is ready to perform an I/O.**
3. **Traps are used to transfer data from a file to a buffer.**
4. **System calls are implemented using traps.**

(green = correct, red = incorrect)

Connect to www.menti.com
Code 7553 5681

Question 5

Which of the following is/are true about interrupts and traps?

1. Upon an interrupt, the CPU enters into kernel mode.
2. An interrupt is a signal from the CPU to a device controller, informing it that the CPU is ready to perform an I/O.
3. Traps are used to transfer data from a file to a buffer.
4. System calls are implemented using traps.

(green = correct, red = incorrect)

Connect to www.menti.com
Code 5402 5022

Agenda

- **Reminder of lecture 1 (quiz)**
- **Processes**

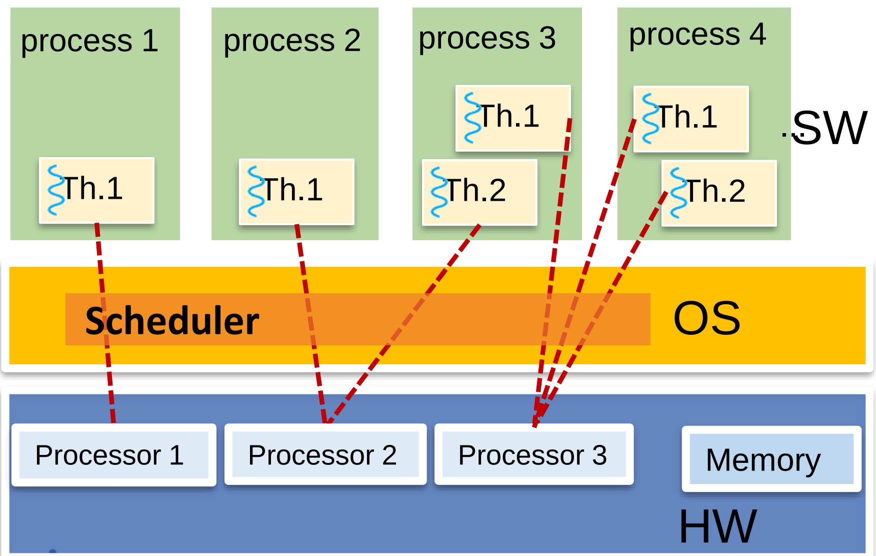
Ingredients of concurrent execution: processes and threads

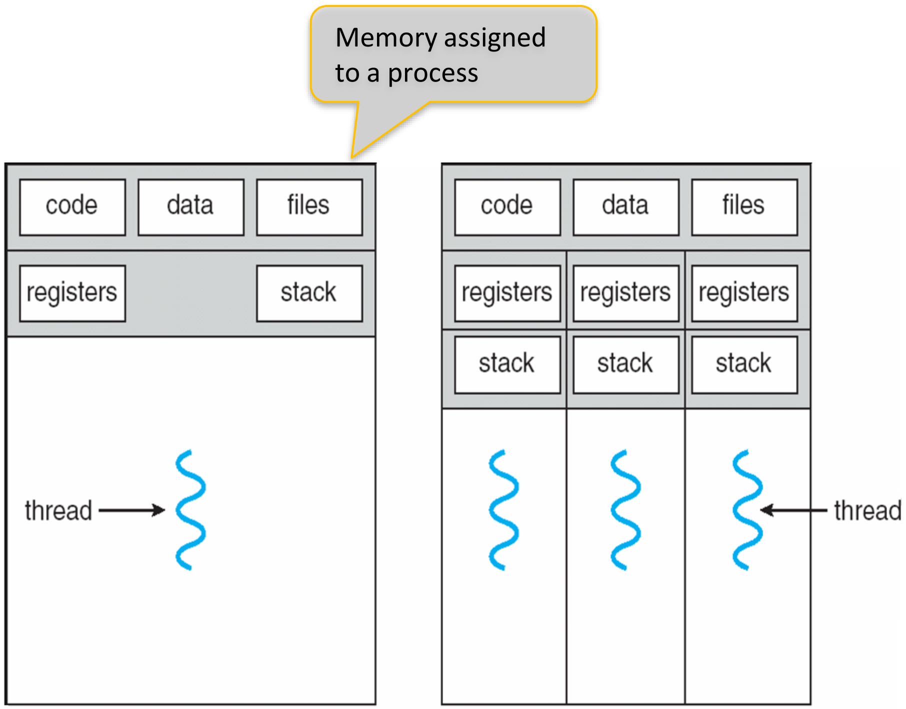
Process

- It is a **program in execution**.
- It has a context of execution
- A process owns resources (has memory and can own other resources such as files, etc.)

Thread

- A **dispatchable unit of work within a process**
- Threads within a process share code and data segments (i.e., share memory address space)





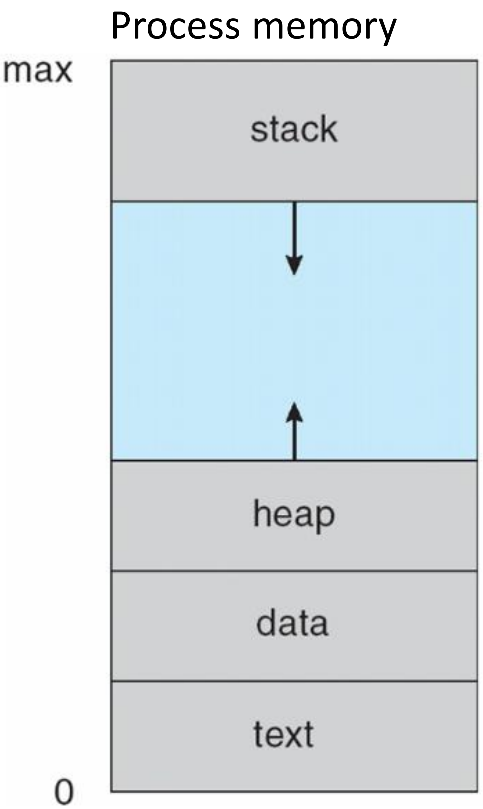
single-threaded process

multithreaded process

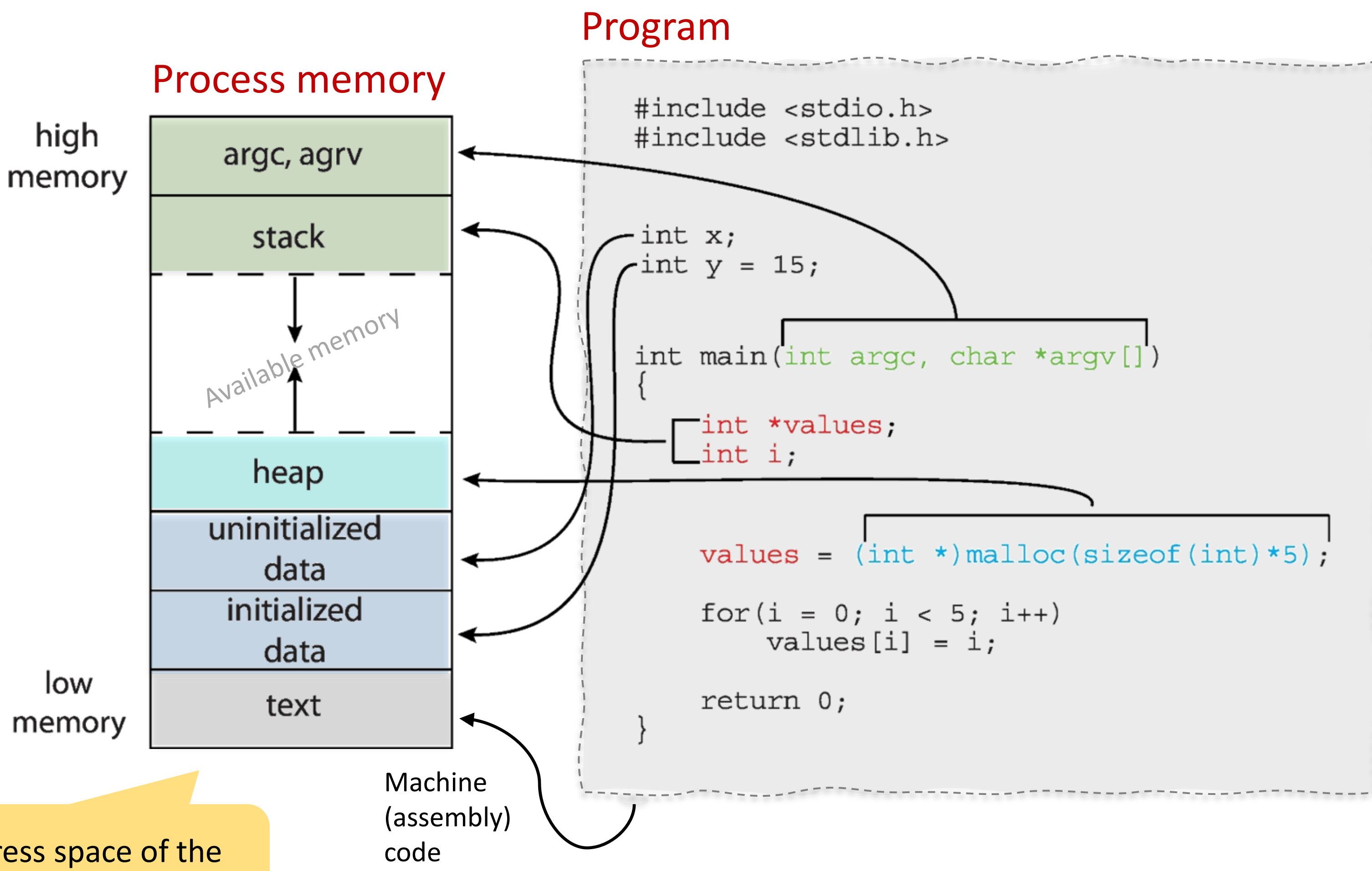
Process

- A process is *a program* in *execution*
  - Several processes can run the same program
  - They all have a different context of execution

- A process is defined by:
  - Text (program code)
  - Stack
    - e.g. local variables, function parameters, return addresses (their sizes known at compile time).
  - heap
    - used to store **data whose size may be unknown** before runtime, e.g. used by a program that reads a file whose size is unknown or uses malloc.
  - data (global variables whose **size are known**)
  - Information about the current (latest) state of execution

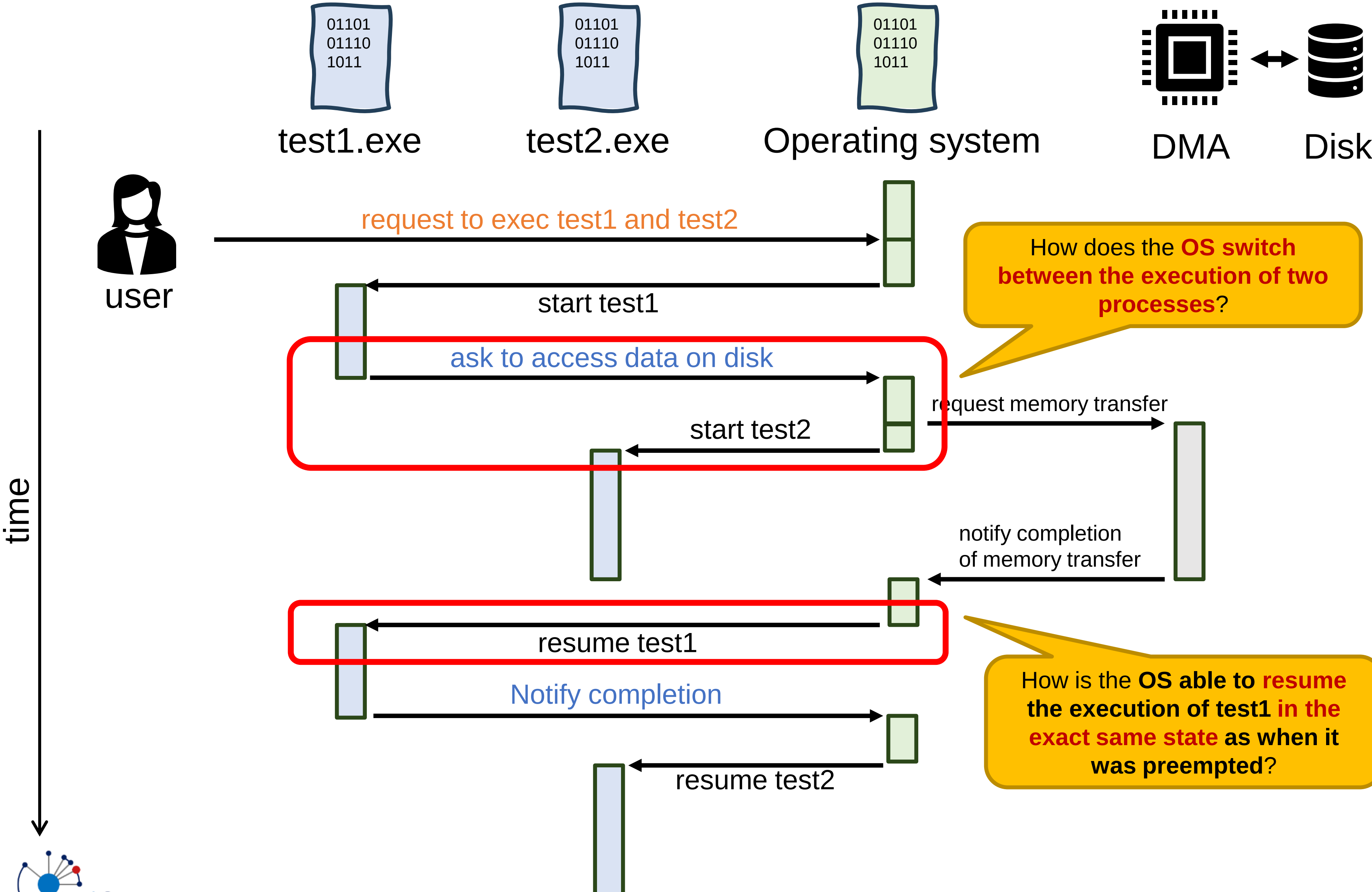


Memory layout of a program in a process



Memory address space of the process running this program

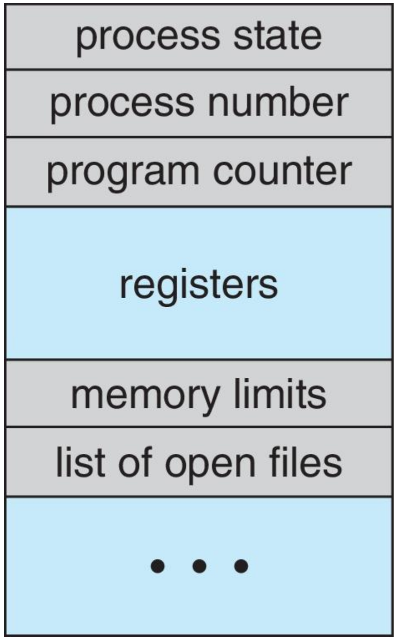
Remember this example?



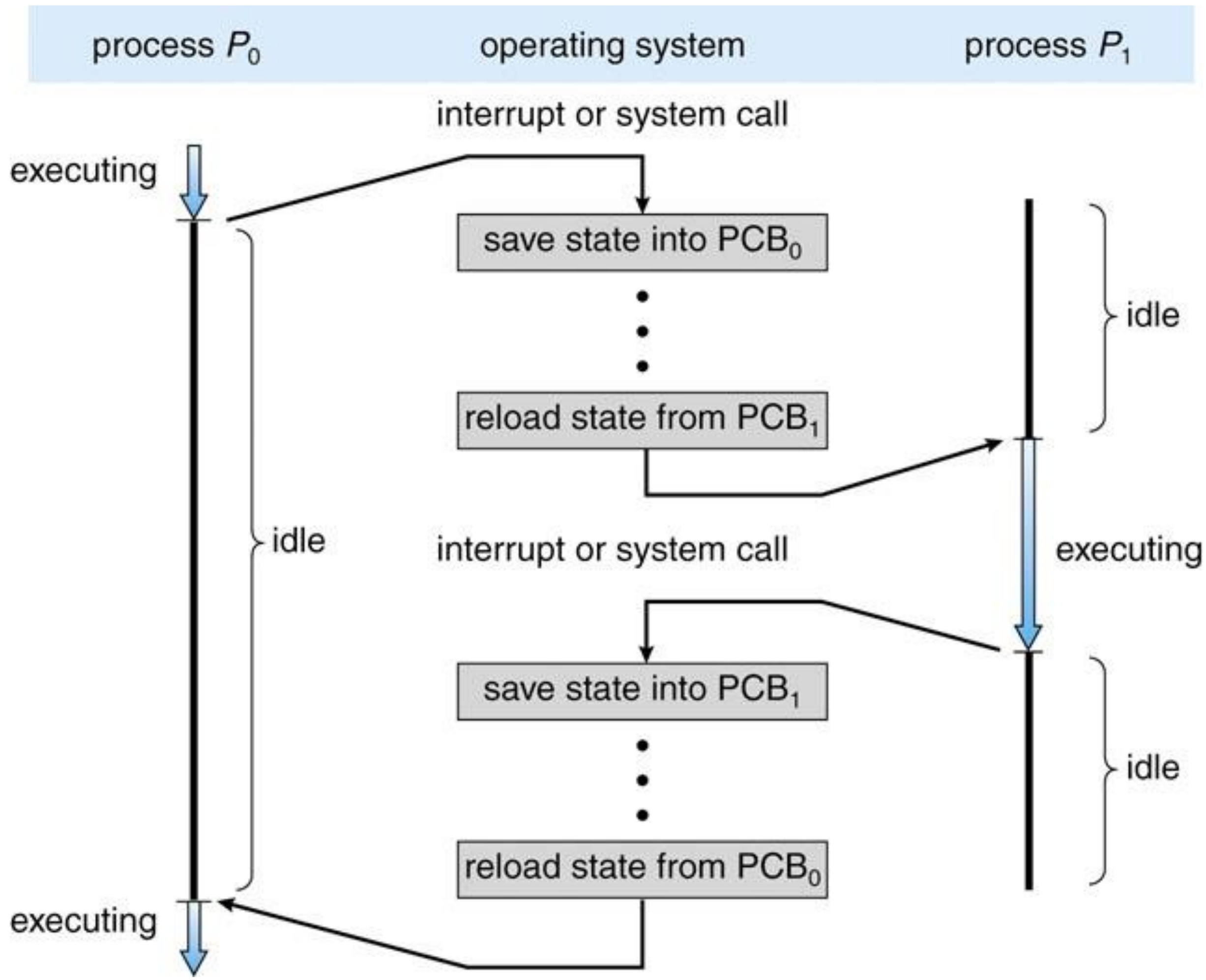
Process control block (PCB)

The **PCB records** information associated with a **process context of execution**

- **Process state** – running, waiting, etc.
- **Program counter** – location of next instruction to execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information**- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



Context switch



Context switch:
Saving the state of a process whose execution is to be suspended, and reloading this state when the process is to be resumed.

Overhead:
(typically) takes a few usecs. → depends on HW

How can we use processes?

Process creation

- **Parent process creates children processes, which, in turn creates other processes, forming a tree of processes**
- **Generally, process identified and managed via a process identifier (pid)**

When a process creates children, it can share its memory or resources (such as files) with them.

Resource sharing options

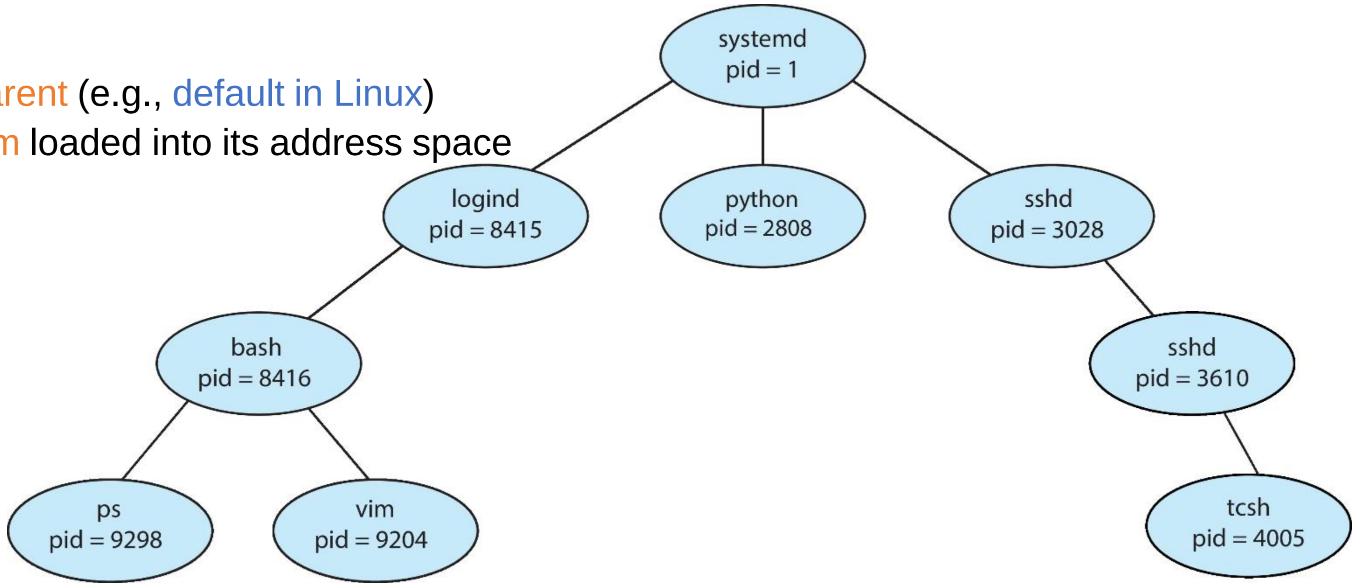
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Execution options

- Parent and children execute concurrently
- Parent waits until children terminate

Address space options

- The child is a copy of the parent (e.g., default in Linux)
- The child has a new program loaded into its address space (e.g., Windows)



Example from Linux

POSIX (Portable Operating System Interface)

- IEEE standard on the API (POSIX is not an OS)
  - Goal: reduce portability effort for applications
  - Many operating systems use POSIX API.

- A system supporting POSIX provides
  - a host language and compiler (often: C)
  - programming interface definition files (e.g., C-header files)
  - programming interface implementation binary or code (e.g., C-libraries)
  - a run-time system (a platform: OS or the like)

We will use the POSIX API to create and terminate processes and threads in this lecture

Example: create new processes

P

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
     perror (“fork”); exit (-1); }

if (child == 0) /* the child */ {
     execlp (“/bin/ls”, “ls”, arg0, arg1, ..., NULL);
     /* this place is reached only in case of error */
     perror (“execlp”); exit (-1);
}
else /* the parent; child == process id of child */ {
     /* do whatever you want, e.g., just return from
     this routine */
     int status;
     wait (&status);

}
......
```

Example: create new processes

fork() creates an identical copy of the caller in a separate memory space;

Only the return value stored in variable ‘*child*’ differs between the two: **0 for the child**, and **child-pid** for the parent

P

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
     perror (“fork”); exit (-1); }

if (child == 0) /* the child */ {
     execlp (“/bin/ls”, “ls”, arg0, arg1, ..., NULL);
     /* this place is reached only in case of error */
     perror (“execlp”); exit (-1);
}
else /* the parent; child == process id of child */ {
     /* do whatever you want, e.g., just return from
     this routine */
     int status;
     wait (&status);

}
......
```

Example: create new processes

fork() creates an identical copy of the caller in a separate memory space;

Only the return value stored in variable ‘*child*’ differs between the two: **0 for the child**, and **child-pid** for the parent

C ⇨ P ⇨

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
      perror (“fork”); exit (-1); }

if (child == 0) /* the child */ {
      execlp (“/bin/ls”, “ls”, arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror (“execlp”); exit (-1);
}
else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from
      this routine */
      int status;
      wait (&status);

}
......
```

Example: create new processes

fork() creates an identical copy of the caller in a separate memory space;

Only the return value stored in variable *'child'* differs between the two: **0 for the child**, and **child-pid** for the parent

C P

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
      perror ("fork"); exit (-1); }

if (child == 0) /* the child */ {
      execlp ("/bin/ls", "ls", arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror ("execlp"); exit (-1);
}
else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from
      this routine */
      int status;
      wait (&status);

}
......
```

Example: create new processes

fork() creates an identical copy of the caller in a separate memory space;

Only the return value stored in variable '*child*' differs between the two: **0 for the child**, and **child-pid** for the parent

execlp() overwrites the calling process with its argument (an executable programme).

This means that code following **execlp** is never reached.

perror() is a generic error printing routine

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
      perror (“fork”); exit (-1); }

if (child == 0) /* the child */ {
```

C

```
      execlp (“/bin/ls”, “ls”, arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror (“execlp”); exit (-1);
}
```

P

```
else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from
      this routine */
      int status;
      wait (&status);

}
......
```

Example: create new processes

fork() creates an identical copy of the caller in a separate memory space;

Only the return value stored in variable '*child*' differs between the two: **0 for the child**, and **child-pid** for the parent

execlp() overwrites the calling process with its argument (an executable programme).

This means that code following **execlp** is never reached.

perror() is a generic error printing routine

```
pid_t child;
......
child = fork();
if (child<0) /* error occurred */  {
      perror ("fork"); exit (-1); }

if (child == 0) /* the child */ {
C ⇨   execlp ("/bin/ls", "ls", arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror("execlp");   exit(-1);
}
P ⇨ else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from
      this routine */
      int status;
      wait (&status);
}
......
```

Create processes with fork() (POSIX)

Multi-shadow-clone jutsu
Naruto

- Every POSIX child process is an **exact copy of the parent process at its creation**
- Child processes become independent after creation

Termination of children: fragment

- Use **exit(status)** to **terminate**

- Functions **wait()**, **waitpid()**
  - wait() blocks until one of the children exit
  - **waitpid()** blocks until a specific child changes its state

```
parent:
    …
    pid_t   child, terminated;
     int status;

    /* blocking wait */
    while (child != wait (&status))      /* nothing */;

Or

    /* or polling wait */
    terminated = (pid_t) 0;
    while (terminated != child) {
            terminated = waitpid (child, &status, WNOHANG);
              /* other useful activities */
            do_something()
    }

    /* both cases: status == 23 */
```

```
child:
    ..... exit (23);
```

More examples: https://www.geeksforgeeks.org/wait-system-call-c/

Termination of children: fragment

- Use **exit(status)** to **terminate**
- Functions **wait()**, **waitpid()**
  - wait() blocks until one of the children exit
  - **waitpid()** blocks until a specific child changes its state

```
parent:
    …
    pid_t   child, terminated;
     int status;

    /* blocking wait */
    while (child != wait (&status))      /* nothing */;
```

Or

```
    /* or polling wait */
    terminated = (pid_t) 0;
    while (terminated != child) {
            terminated = waitpid (child, &status, WNOHANG);
              /* other useful activities */
            do_something()
    }

    /* both cases: status == 23 */
```

This parameter is used to continue without hanging if the child "Child" did not finish yet.

```
child:
    ..... exit (23);
```

More examples: https://www.geeksforgeeks.org/wait-system-call-c/

Process termination

- Process may ask the operating system to delete itself (**exit**)…
  - exit() returns status value from child to parent (if the parent was "**waiting**")
  - After termination, the process' **resources** are taken away by OS

- … or **parent** may terminate execution of **children** processes (**abort / kill**)
  - The child has **exceeded allocated resources**
  - The task assigned to child is **no longer required**
  - If the **parent is exiting**
    - Some operating systems do not allow the child to continue if its parent terminates, in which case all children are terminated - **cascading termination**
    - Some operating systems **attach the orphan children to a 'grandfather' process** (e.g. *init*)

- If no parent is **waiting** (did not invoke **wait()**) the child process is a **zombie**
- If parent **terminated** without invoking **wait()**, the child process is an **orphan**

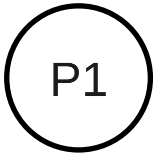
Exercises

Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```

How many processes are created, including the initial parent process?

Process creation

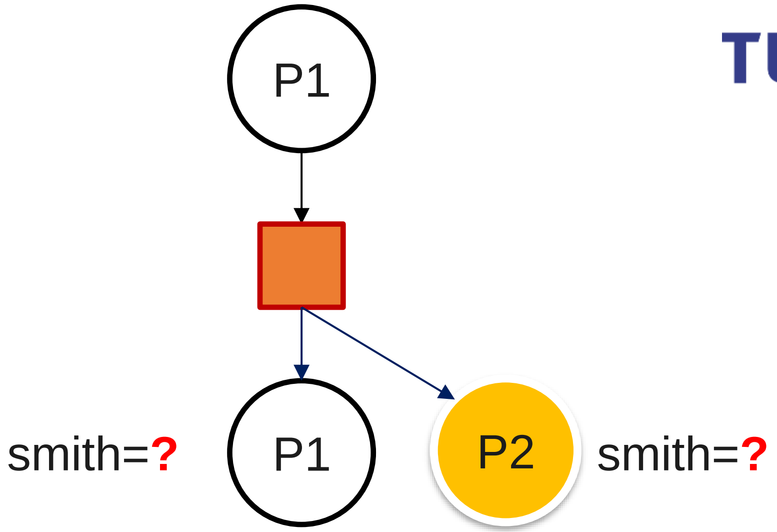


Answer: 5

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```

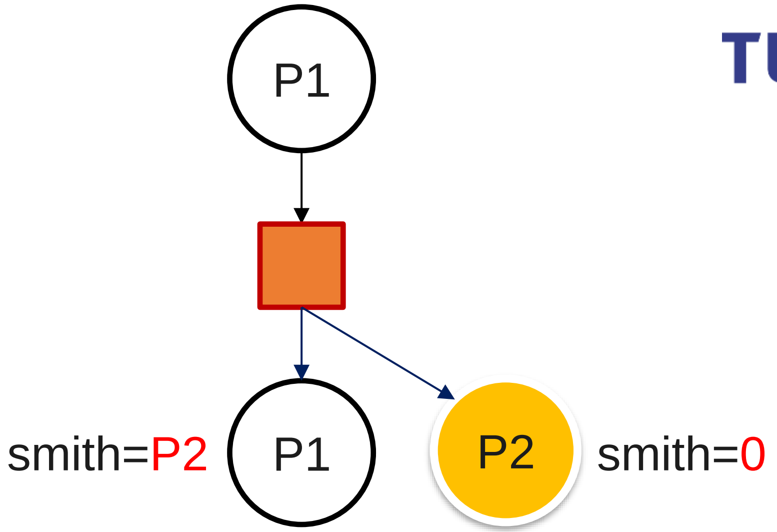
Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```



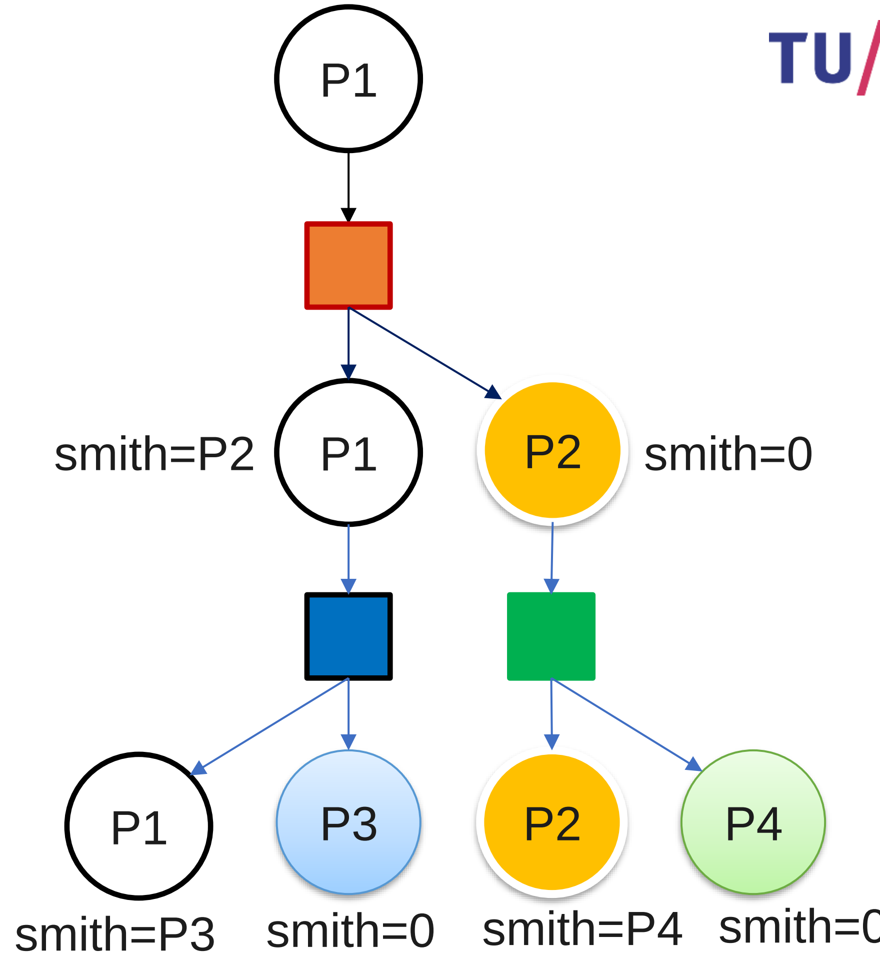
Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```



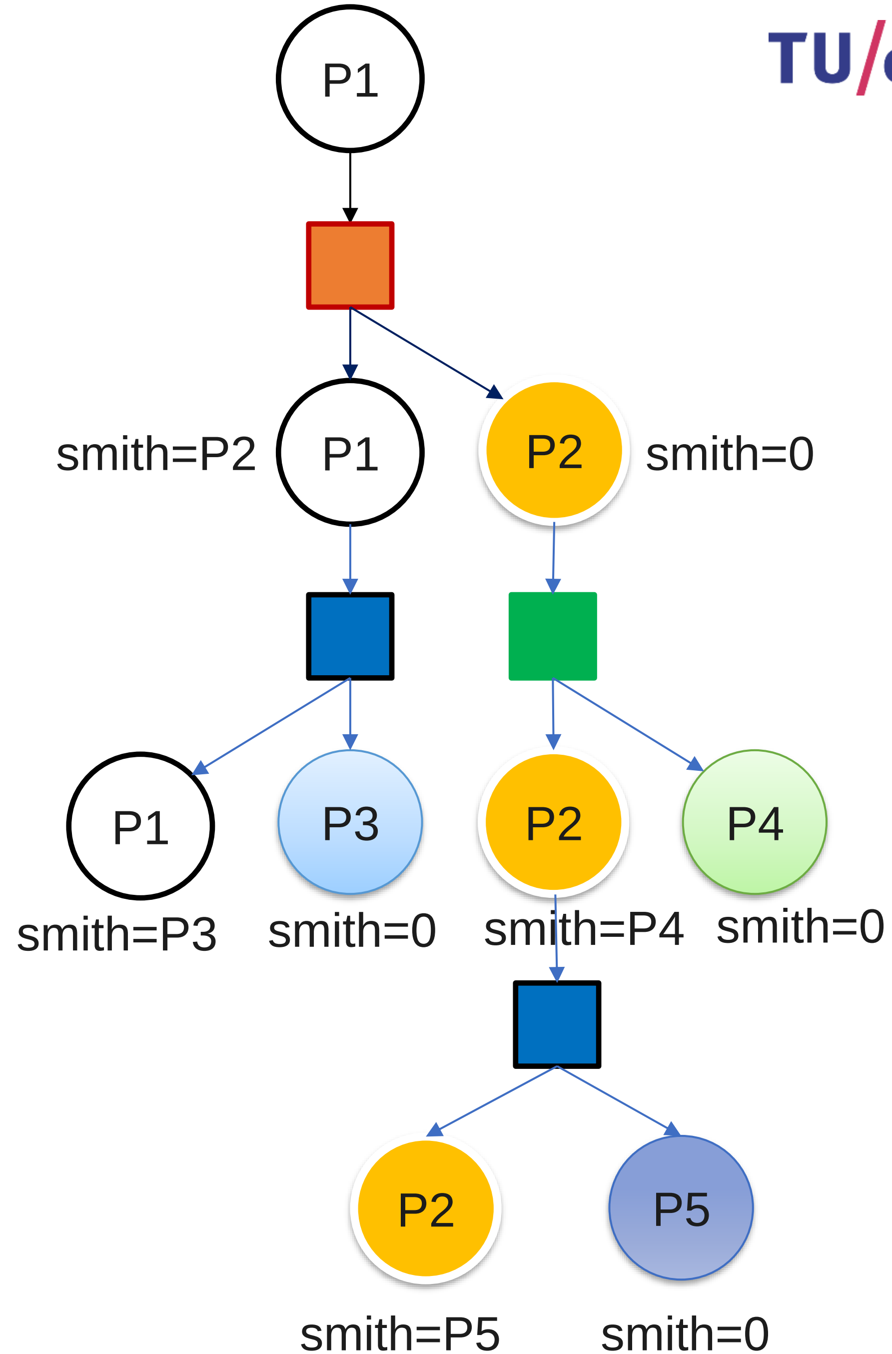
Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```



Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```



Process creation

```
#include ...
#include ...

int main() {
    pid_t smith;
    smith = fork( );
    if (smith == 0) { smith = fork( );}
    if (smith > 0) { smith = fork( );}
    /* do something useful */
    return 0;
}
```

There are two issues in the code. Can you spot them?

Process execution

```
int i = 0;
int main() {
    pid_t id;
    id = fork( );
    if (id == 0) {
        exit(0);
    }
    id = fork();
    if (id > 0) { fork(); }
    i++;
    /* do something useful */
    return 0;
}
```

How many processes are created (including the initial parent process)?

How many processes **execute the "i++" instruction**?

What is the **final value of** the variable ***i*** **in the last process that reaches "return 0"**?

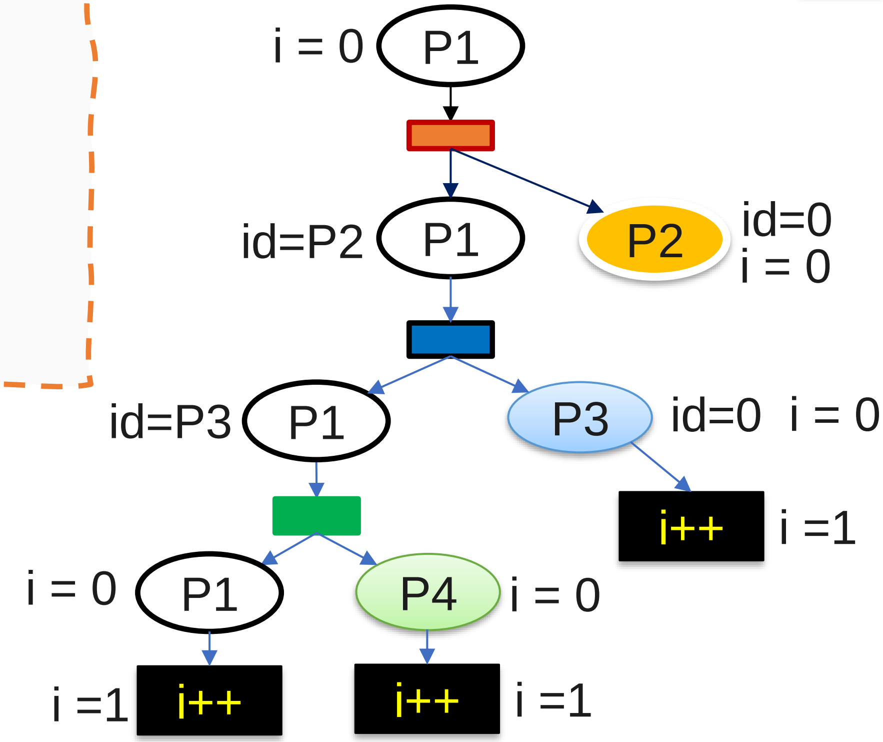
Process execution

```c
int i = 0;
int main() {
    pid_t id;
    id = fork( );
    if (id == 0) {
        exit(0);
    }
    id = fork();
    if (id > 0) { fork(); }
    i++;
    /* do something useful */
    return 0;
}
```

How many processes are created (including the initial parent process)? **4**

How many processes **execute the "i++" instruction**? **3**

What is the **final value of** the variable ***i*** **in the last process that reaches "return 0"**? **1**



Reminder

Remember: the content of the preparatory material is also part of the exam
(e.g., process state, inter-process communication, etc.)

Summary

- **Ingredients of concurrent execution: processes and threads**
  - **Process**
    - a **program in execution**.
    - has a context of execution
    - owns resources
  - **Thread**
    - **dispatchable unit of work within a process**
    - **shares code and data**

- **Creation/deletion of process**

Next week

- Lecture on threads and scheduling

- Preparation:
  - Reference book (~11 pages)